16 User-defined functions

16.1 Adapting Perl to our own tasks

In chapter 7 we saw that Perl includes dozens of built-in functions, such as `sqrt()`, `substr()`, `rand()`, and we examined a few of them. But often we need our program to execute some specific function that does not feature on the list of built-in functions, perhaps because it relates too closely to the particular task we are working on. In that case, we will need to create our own *user-defined function*.

For instance, in our code-chunk (18) to read in a file of county population data, one of the actions was to replace any sequence of one or more whitespace characters internal to a string with a single underline character, so that a string of the form, say,

```
Cambridgeshire and   Isle of Ely
```

is turned into:

```
Cambridgeshire_and_Isle_of_Ely
```

In our program this was handled by a single line, line 18.14:

```
$name = join("_", split(/\s+/, $name));
```

But in a larger program, which perhaps has to read various kinds of data in from several different files, there might be numerous situations where we want to make names more “machine-friendly” by carrying out this same routine of changing internal whitespace to single underline characters. We could copy 18.14 at every point in the program where this routine needs to be executed, replacing `$name` with whatever variable the routine needs to be applied to at the point in question. But 18.14 is a complicated thing to type out (even in its newer form – in its original form as line 12.14 of program (12) it was even more fiddly); it is inefficient and invites errors to repeat such details at different places in a program.

What we need to do is define a function, call it `despace()`, which takes a string as argument and does to it what 18.14 does to the string `$name`. Once the `despace()` function is defined, we can simplify (18) by replacing 18.12 (which assigns the value of the temporary variable `$1` to `$name`) and 18.14 (which, as we have seen, replaces whitespace by underlines in `$name`) with the single line:

```
$name = despace($1);
```

and, wherever else in a longer program we need to alter character strings in that particular way, we can do so using a similarly concise statement.

16.2 The structure of a user-defined function

So how do we define the `despace()` function? Like this:

(19)

```
1   sub despace
2     {
3     my $p = shift(@_);
4     $p = join("_", split(/\s+/, $p));
5     return($p);
6     }
```

The keyword `sub` stands for "subroutine"; the terms "function" and "subroutine" are interchangeable in Perl.[26]

It is logical to keep function definitions separate from the main sequence of statements that a program executes when it is run, so commonly a function definition like this would appear either at the beginning or at the end of the program. But Perl does not actually care where a user-defined function definition appears; whenever Perl reaches a statement which *calls* that function – that is, a statement such as `$name = despace($1)`, whose meaning depends on the definition of `despace()` – then Perl will find the relevant function definition and use it, so long as it exists in the program file somewhere.

What the `despace()` definition is essentially saying is this: "Whatever the argument of this function is actually called, for the purposes of defining the function it will be called `$p`. Now `$p` is changed in the way that line 18.14 changed the string `$name`. And after that has been done, the new version of `$p` is delivered back to the code which called the function, as the result *returned* by the function call." The keyword `return` identifies what follows it as the result of that function call.

The only new complication in this relates to line 19.3, `my $p = shift(@_)`. Bear in mind that at the time we write the function definition, we don't know what its argument will be named when the function is used. In (18) the string which had its spaces changed to underlines was called `$name`, but on another occasion we might want to apply `despace()` to a string called `$a123`, or `$broccoli`, or anything. So we need an arbitrary name internal to the function definition to stand for the real name that appears in the calling code – any arbitrary name would do, I happened to choose `$p`. Line 19.3 is what tells Perl that the role of `$p` in the function definition is to represent the argument.

Perl does this in a more roundabout fashion than some programming languages. If Perl were more "ordinary" in this respect, line 19.1 might run `sub despace($p)` – announcing the name to be used for the argument at the same time as the function name `despace` is introduced. In Perl that does not work. Instead, whenever any function is called, whatever argument(s) it is called with are assigned to a special array name `@_`. (Remember that a function may have more than one argument; this applies to user-defined functions as well as to built-in functions.) In the case of `despace()` there is only one argument, so `@_` will be a one-element array; and `shift()` will give us the first element (in this case, the only element) of the `@_` array. Line 19.3 assigns that argument to a name `$p`, under which name we are free to manipulate it.

The remaining feature of line 19.3 which needs explanation is the word `my`. This word before a variable name makes the variable *local* rather than *global*. If we are writing a large program, as well as many lines of "top-level" code outside any subroutine, there may be numerous different subroutines – indeed it is likely that some complex subroutines will themselves call lower-level subroutines, and so on down. While we are defining a subroutine needed for some quite separate aspect of the program, we will probably have forgotten all about the details of `despace()`, and we could easily decide to use the variable name `$p` for some unrelated purpose in that other subroutine. Unless we explicitly declare `$p` as local to `despace()`, the same name `$p` appearing in another subroutine will refer to the same variable, and things done in one subroutine might interfere with what is intended to happen in the other. Variables are global by default, but we want `$p` in `despace()` to be local, so that it is treated by Perl as independent of any variables that happen to share the same name in other parts of the program. The word `my` achieves that.

Line 19.3 declared `$p` as local at the same time as `$p` was assigned a value, but a `my` statement can be a separate statement; we could have written:

```
my $p;
$p = shift(@_);
```

If our subroutine needs several local variables, we can declare them all in a single line within brackets:[27]

```
my ($p, $q, $r);
```

The *scope* within which a variable is local does not have to be a subroutine, but that is probably the most usual way to use `my` and the only one we shall discuss.

Returning to our definition of `despace()`: once 19.3 has defined `$p` as standing for the argument, 19.4 then applies exactly the same substitution operation to `$p` as line 18.14 applied to `$name`. Finally, the word `return` in 19.5 defines what the function delivers to the calling code as its result – in this case, `$p` as it exists after 19.4 has applied to it.

So code-chunk (18) will now be replaced by (20), in which the main part of the program contains a line 20.12 which calls the function `despace()` with the string `$1` captured by line 20.6 as argument, and the result of applying `despace()` to `$1` becomes the value of `$name`:

(20)

```
1   open(INFILE, "<../data/pops/countyData.txt")
      or die "Cannot open input file\n";
2   $i = 0;
3   while ($a = <INFILE>)
4     {
5     if ($a !~ /\S/) {;}
6     elsif ($a !~ /^\s*(\S.*\S)\s+(\d+)\s*$/)
7       {
8       die "bad input line:  $a\n";
9       }
10    else
11      {
12      $name = despace($1);
13      $population = $2;
15      $population *= 1000;
16      $counties[$i][0] = $name;
17       $counties[$i][1] = $population;
18       ++$i;
19      }
20    }
21  close(INFILE);

22  sub despace
23    {
24    my $p = shift(@_);
25    $p = join("_", split(/\s+/, $p));
26    return($p);
27    }
```

16.3 A second example

The function `despace()` is one that applies to strings. As further practice with the concept of user-defined functions, let's now define a function that works with numbers.

In chapter 7 we saw that the built-in function `int()` turns decimals into whole numbers in a crude fashion, which does not give the nearest whole number if the fractional part of the argument is half or more. It could often be useful to have a function `round()`, which always takes a decimal to the nearest whole number, rounding up or down as appropriate. This function is easy to define.

If you think about it, one way to get the right answer would be to add one-half (0.5) to the number we want to round, and then apply the built-in `int()` function to the result. We want e.g. 3.14 to be rounded to 3: 3.14 plus one-half is 3.64, and `int(3.64)` is 3; but we want e.g. 3.75 to be rounded to 4: 3.75 + 0.5 = 4.25, and `int(4.25)` gives 4. Well, actually, this is not quite the whole story, because we haven't taken negative numbers into account yet. If the argument to `round()` is negative, we want to *subtract* one-half before applying `int()`, because we want `round(-3.14)` to be –3 and `round(-3.75)` to be –4.

So here is a function which will give the correct result with both positive and negative arguments:

(21)

```
1  sub round
2    {
3    my $n = shift(@_);
4    return($n >= 0 ? int($n + 0.5) : int($n - 0.5));
5    }
```

Line 20.4 uses the `?:` construction to ask "Is `$n` positive? If yes, return the result of applying `int()` to `$n` plus 0.5; if no, return the result of applying `int()` to `$n` minus 0.5."

16.4 Multi-argument functions

Functions such as `despace()` or `round()`, which take a single argument, are the simplest case. To broaden our grasp of user-defined functions, let's look at one taking several arguments. Perhaps we are interested in checking the average length of collections of words used in some circumstance, and we want to count only wholly-alphabetic words, not "words" composed partly or wholly of numerals. So we want to define a function `aveWordLen()` which gives results as follows:

```
print aveWordLen("new", "sale", "special", "executive");
```

```
5.75
```

```
print aveWordLen("leading", "99p", "bargain");
```

```
7
```

(The words *leading* and *bargain* both have seven letters, and *99p* will be ignored because it contains digits.)

Notice that `aveWordLen()` does not merely take multiple arguments; it can take any number of arguments on different occasions. Since an array can be any length, the `@_` system means that this is no special problem.[28]

Here is a definition for `aveWordLen()`:

(22)

```
1   sub aveWordLen
2     {
3     my $total = 0;
4     my $N_words = 0;
5     my $word;
6     foreach $word (@_)
7       {
8       if ($word !~ /\d/)
9         {
10        ++ $N_words;
11        $total += length($word);
12        }
13      }
14    return($total / $N_words);
15    }
```

Line 22.6 assigns each string in the argument array successively to the local variable `$word`; 22.8 checks that `$word` contains no digit, and (provided it passes the check) lines 22.10–11 increment `$N_words` by one and `$total` by the length of `$word`. Line 22.14 defines the result as the total of all word lengths divided by the number of words.[29]

16.5 Divide and conquer

It is clear that if a function like `aveWordLen()` is needed at several places in a program, it is much better to define it once and call it by name whenever it is needed, than to copy a long chunk of code such as the one above into all the different places where it is needed. It is not just a matter of reducing the overall length of the code – that in itself does not matter much (though the more lines, the more opportunity for typing mistakes). But more important is the principle of "divide and conquer". When a program is required to execute a complex range of processes, the programmer is much more likely to be able to get it right if the overall functionality can be broken down into many small pieces, each of which can be coded independently of the others – so that only a limited number of issues have to be held in the programmer's head at any one time. User-defined functions and local variables are a large part of the machinery which Perl, and other languages, use to facilitate the "divide and conquer" approach.

Consequently, an important part of learning to program well is learning to notice cases where parts of a longer process can be separated out and coded as subroutines. It is rarely a good idea for a program to contain a lengthy "main" section (the part of the code which the machine begins to execute when the program is run, outside all the `sub`'s). Good, robust software is commonly articulated into many separate functions, so that a fairly short main section contains several function calls (and the functions will often call further functions, and so on).

16.6 Returning a list of values

Each function we have defined so far has returned a single item (a scalar). But we can also define a function to return a list. Let's define a function `maxima()` which uses our reformatted file of county population data to identify both the largest county population and the longest county name. This is perhaps not a very realistic task in practice – why would we want to investigate both of those two issues simultaneously? – but it makes a good exercise. (Tasks that are useful for teaching programming concepts often are unrealistic.)

We'll say that in the main part of our program, outside the subroutines, there will be lines which simply ask for the two maximum figures and incorporate them in messages to be printed out, like this:

(23)

```
@figures = maxima();
print ("Largest county pop. is $figures[0]\n");
print ("Longest county name is $figures[1]\n");

Largest county pop. is 7,914,000
Longest county name is Huntingdonshire_and_Peterborough
```

The data will be taken from the reformedData file created by program (12); and all the work of reading these data in and working out which entries are the respective maxima will be delegated to the `maxima()` function.

Remember that in the reformedData file each line contains a population figure with commas grouping digits into threes, followed after a tab by a county name, e.g.:

```
353,000 Cornwall
```

We shall be doing arithmetic with the population figures, to discover which is largest, so `maxima()` will need to remove the commas before comparing one figure with another (we know that Perl does not recognize a string such as `353,000` as a number). We can handle this with a further user-defined function, `decomma()`, which takes a string as argument and returns the string with any commas stripped out and the gaps closed up. And when returning the maximum population as a figure ready to be displayed to the user, it would be nice to have the commas put back in (as in the output shown in italics above); so we shall write another function, `recomma()`, which will use the final version of line 12.16 (see section 11.3 above) as a routine for inserting commas in the correct places.

Here are the three functions. (I have left a blank line between them, and in typing out a program one normally would leave some space between separate subroutines, simply for readability; Perl does not care whether the blank line is there or not.)

(24)

```
1   sub maxima
2     {
3     open(INFILE, "../data/pops/reformedData.txt") or
        die("Can\'t open reformedData.txt\n");
4     my($longestName, $largestPop, $line, $name, $pop);
5     $longestName = "";
6     $largestPop = 0;
7     while ($line = <INFILE>)
8       {
9       chomp($line);
10      if ($line !~ /^(\S*)\t(\S*)$/)
11        {
12        die("input line in unexpected format: $line\n");
13        }
14      $pop = decomma($1);
15      $name = $2;
16      if (length($name) > length($longestName))
17        {
18        $longestName = $name;
19        }
20      if ($pop > $largestPop)
21        {
22        $largestPop = $pop;
23        }
24      }
25    close(INFILE);
26    return(recomma($largestPop), $longestName);
27    }

28  sub decomma
29    {
30    my $numeral = shift(@_);
31    $numeral =~ s/,//g;
32    return($numeral);
33    }

34  sub recomma
35    {
36    my $popFig = shift(@_);
37    while ($popFig =~ s/(\d)(\d{3})($|,)/$1,$2$3/)
38      {;}
39    return($popFig);
40    }
```

The function `maxima()`, which is a function taking no arguments, begins (24.5–6) by setting the variables `$longestName` and `$largestPop` to the shortest possible name (the length-zero "null string") and smallest possible population (zero) respectively, and then looks at the name and population on each line of the input file. Since we are taking our input data from a file we created ourselves, it should be safe to assume that each line has the numeral + tab + name format – but just in case the reformedData file has somehow become corrupted, 24.10 arranges for the program-run to terminate with an appropriate error message if a line with a different format is encountered. For each population figure read in, 24.14 in the `maxima()` function calls the `decomma()` function to change it into a form it can do arithmetic with. Whenever `maxima()` finds a name or a population larger than the previous maximum found, the new name or population replaces the current value of `$longestName` or `$largestPop` respectively as current maximum (24.16–23). The `maxima()` function returns the overall maxima as a list with two members, within which the population maximum is in the form produced by the `recomma()` function; in (23), within the main part of the program, this two-member list is assigned as the value of an array `@figures`.

Thus in (23) the main part of the program calls `maxima()`, which in turn calls `decomma()` and `recomma()`; the return values of `decomma()` and `recomma()` contribute to calculating the return value of `maxima()`, and that return value in turn contributes to executing the statements in (23). This is how real-life Perl programs will typically be structured. The main part of the program, outside any `sub` definition, may be quite short, but it will contain function calls; the subroutines they call will often themselves contain calls to further functions, and there may be further levels below levels of user-defined functions until one eventually gets down to functions that use only built-in Perl constructions.

16.7 “Subroutines” and “functions”

Those readers who have learned other programming language(s) before encountering Perl might be surprised that Perl uses the keyword `sub` to define functions, and that I have been using the terms “function” and “subroutine” interchangeably – as mentioned earlier, for languages other than Perl, subroutines and functions are two different things. A subroutine *does* something – it carries out an action, for instance it changes the values of its arguments in some systematic way, displays the arguments in some format, or the like; a function calculates a value and returns it for use by another part of the program, commonly without changing anything. And this difference in purpose is reflected, in many programming languages, by different notation conventions.

In Perl there is no difference between subroutines and functions. All the user-defined functions we have looked at so far contained a `return` statement returning a value to the part of the program which called the function; but a Perl function does not have to contain a `return` statement. Here is a function which takes a string as argument and prints out its words (elements separated by whitespace) on separate lines, in other words it carries out an action rather than returning a value:

(25)

```
1   sub printlnbyln
2     {
3     my $string = shift(@_);
4     my @words = split(/\s+/, $string);
5     my $word;
6     foreach $word (@words)
7       {
8       print($word, "\n");
9       }
10    }
```

A program containing this function definition could have a statement like:

```
printlnbyln("the quality of mercy");
```

```
the
quality
of
mercy
```

Since `printlnbyln()` contains no `return` statement, one would not normally use a statement like:

```
$a = printlnbyln("the quality of mercy");
```

(In fact Perl is organized so that even subroutines without `return` statements do deliver return values; but this is one of the Perl wrinkles that is best left to the geeks. Learners are advised only to use return values from defined functions where the return values are explicitly identified as such by `return` statements.)

We saw in chapter 9 that while most Perl built-in functions return values, some, such as `chomp()`, exist mainly in order to change their arguments. Likewise a code block headed by a `sub` line can be designed to carry out an action, it can be designed to return a value, or it can be designed do both of these. There is no distinction between subroutines and functions, so it is reasonable to use `sub` as a keyword for user-defined "functions".